Hackerearth

EXL™ Hackathon
Cloud Agnostic File Storage System

BY : Gana Sai Praveen. K (Team: Gana_Rko)

Problem Statement:

The current solution processes backend requests and generates reports in .pdf, .csv, or .pdm file formats. These reports are then uploaded on an on-premises shared file server. Since the solution is now hosted on cloud, a file-server-based model is not the optimum solution for file storage.

Though transitioning file storage from on-premises to cloud can be achieved easily, clients have their choices of cloud provider, which requires customization for every transition. A cloud-agnostic storage solution can address this problem.

Can you build a solution that is agnostic of the cloud provider for storing and retrieving files, i.e. one solution that supports storage and retrieval of files from AWS S3 bucket, Azure Blob storage, or Google Cloud storage.

Key takeaways from Problem Statement

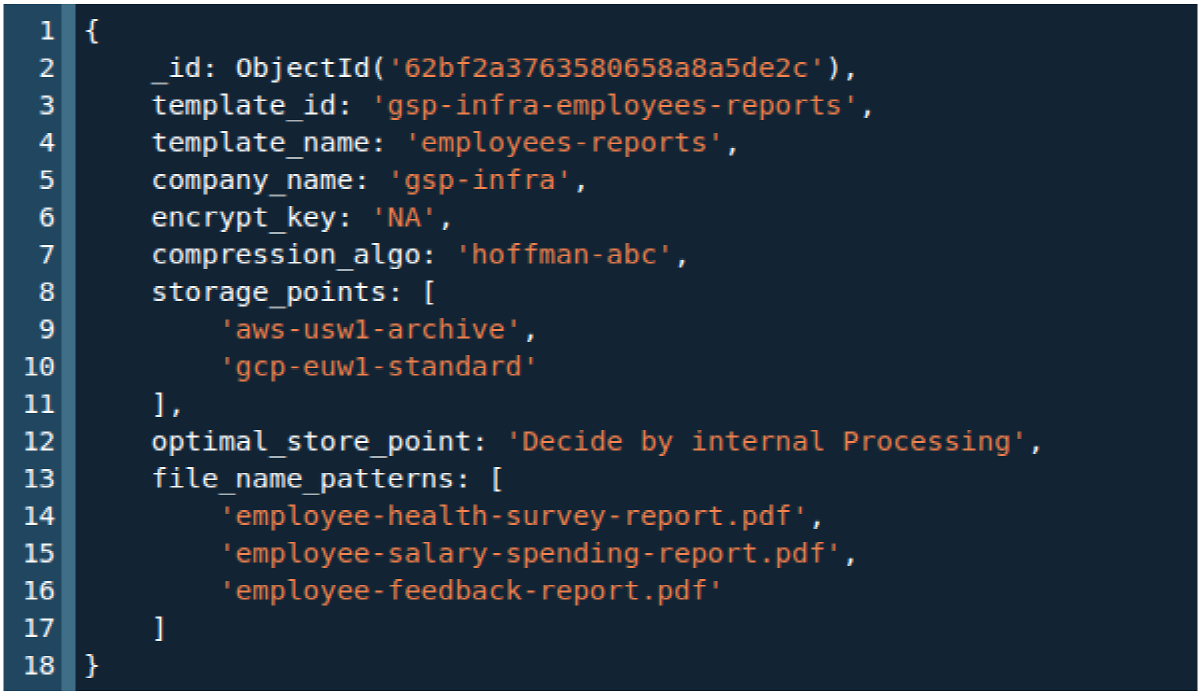
- The Problem **isn't about having storage system in cloud** (highlighted in red 🟥)
- The soul of the problem is about **providing clients the flexibility on how we store** their data /reports. (highlighted in green 🟩)
- Observing the Storage Options, The solution would be better, if we can take advantage of the features these Cloud providers offer on Global scale.
- So this solution will focus on the following aspects:
  - Customization in the way we store client reports
  - Accessibility of the reports for the Clients
  - Affordability of the reports for the Clients
  - Security of The Stored Data
  - Budget Implementation of the Solution (for us)

Solution

- The Solution consists of two micro services one is called a template Server and the Other one is Report Server (written in Django)
- Here's the description of each of them

Template Server

- This is microservice backed up with a mongoDB
- This will enable our Clients to define a template with various properties on how a report can be stored, they define,modify templates here and specify on what sort of their reports this template should be considered while storing the report on cloud providers.
- Here’s a sample template from the prototype. This mongoDB object represents what fields a client can specify in a template and they will be used while storing files of clients choice.



```
{
    _id: ObjectId('62bf2a3763580658a8a5de2c'),
    template_id: 'gsp-infra-employees-reports',
    template_name: 'employees-reports',
    company_name: 'gsp-infra',
    encrypt_key: 'NA',
    compression_algo: 'hoffman-abc',
    storage_points: [
        'aws-usw1-archive',
        'gcp-euw1-standard'
    ],
    optimal_store_point: 'Decide by internal Processing',
    file_name_patterns: [
        'employee-health-survey-report.pdf',
        'employee-salary-spending-report.pdf',
        'employee-feedback-report.pdf'
    ]
}
```

Back

- Fields in the Object:
    - **_id:** id generated by mongoDb per Object
    - **template_id:** uniques id assigned by our template server for one template
    - **company_name:** Name of the client/Company owning this report
    - **encrypt_key:**
        - Security is a serious concern, while storing reports. As some reports might contain proprietary findings/stats/data too. All major Cloud providers provide encryption (managed by them, with our encrypt key no encryption at all etc..).
        - Here in this field clients can provide their own encryption key to encrypt/decrypt his report data. This allows him to use both symmetric and asymmetric encryptions for the generated report, depending on their need of confidentiality.
        - For symmetric encryption they can just provide the key and in case of asymmetric encryption they can provide a public key in which case no-one but only they can decrypt the report after it got encrypted and stored.
    - **compression_algo:**
        - As clients will be billed in one way or the other for their reports storage, that we manage. It's important to optimize the cost of storage for them too.
        - One way to save bucks is, via compressing files, while storing them, but usually some cloud-providers provide this feature and some don’t.
        - But what if we have an algorithm which compresses files we generate to much smaller size?. The problem with cloud provides is that due to their large infrastructure and global presence, they cannot implement use-case specific procedures effectively, at least not at the same pace as the development of new features in the real world.
        - Thus this particular field will help us to implement use case specific compression to make storage options more affordable to our clients.

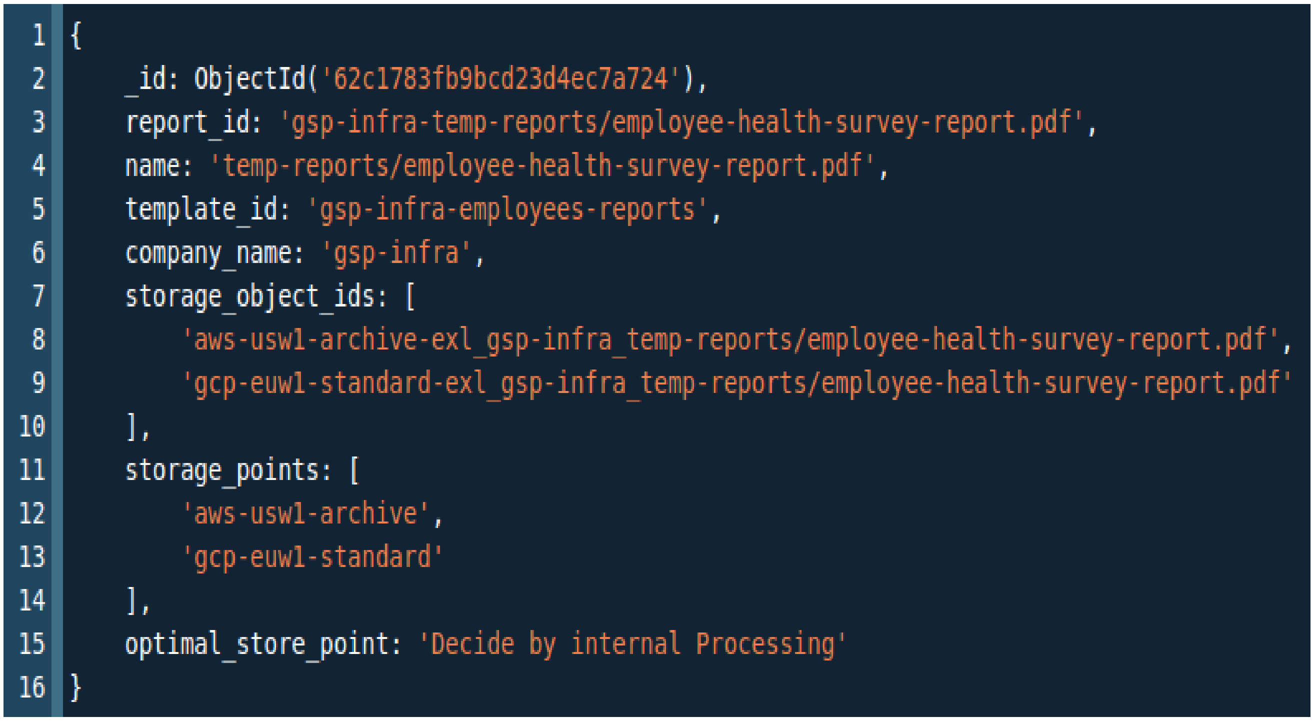
- **storage_points:**
    - Each entry here will follow the format
      Cloud-Provider - Storage Bucket Location - Type of Storage

      Example: ***gcp-euw1-standard***

    - **Cloud-Provider:** Cloud Provider they choose to go with, this part enables them to store reports across multiple cloud providers, Cloud Agnostic isn't it !!?? :-)
    - **Bucket Location:** As we now enable clients to choose from multiple storage providers, it'll be more useful if we also allow them to choose where to store their reports (this matters for accessibility).
    - **Type of Storage:** Almost all cloud providers have different types of storage (SSD, HDD etc) with different options, as clients will be aware of the way they use these reports they can use this aspect to optimize their spending.(this matters for affordability).
    - **Lifespan:** Some reports might not be needed forever, so this can set a max stay period for a report in a cloud bucket. After this period of time this report object can be moved to cheaper storage or removed based on clients requirements. (not implemented in proto type)
- **file_name_patterns:**
    - It'll be a huge burden for our client, if he was asked options to customize storage options for every report we generate
    - So to bypass this trouble, we included this field, where clients can simply define their report_names or patterns in the template once for all.
    - Report Server while uploading the report will check for the template with matching report name and apply all customizations from the above described options while storing that file.
- **optimal_store_point:**
    - This field will help report servers to ease out on computation while storing reports over multiple storage providers. (not operational in prototype)

Report Server

- This is microservice backed up with the same mongoDB as Template Server
- This server will be coupled with our report Generator Service to store files in the cloud.
- This micro service creates a metadata object in mongoDB for each report it stores.
- Here's a sample template from prototype



```
{
    _id: ObjectId('62c1783fb9bcd23d4ec7a724'),
    report_id: 'gsp-infra-temp-reports/employee-health-survey-report.pdf',
    name: 'temp-reports/employee-health-survey-report.pdf',
    template_id: 'gsp-infra-employees-reports',
    company_name: 'gsp-infra',
    storage_object_ids: [
        'aws-usw1-archive-exl_gsp-infra_temp-reports/employee-health-survey-report.pdf',
        'gcp-euw1-standard-exl_gsp-infra_temp-reports/employee-health-survey-report.pdf'
    ],
    storage_points: [
        'aws-usw1-archive',
        'gcp-euw1-standard'
    ],
    optimal_store_point: 'Decide by internal Processing'
}
```

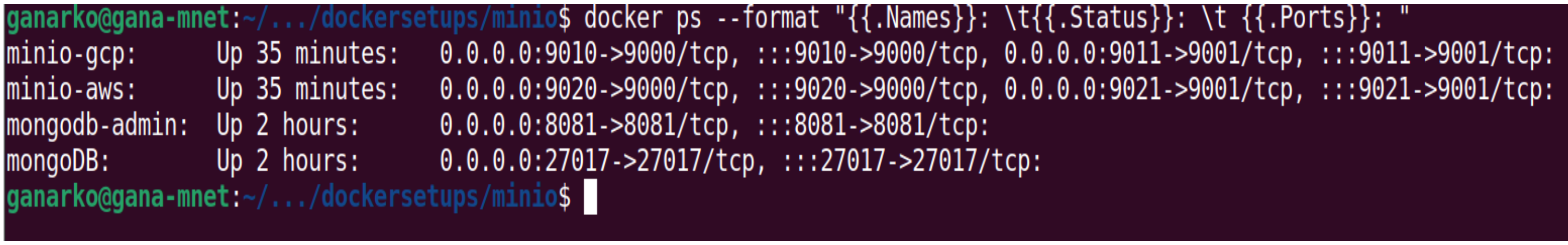
- Fields in the Report metadata Object:
    - **_id:** id generated by mongoDb per Object
    - **report_id:** uniques id assigned by our report server for each report
    - **name:** Name of the report
    - **template_id:**
        - ID of Template used while storing this report, as we'll require some properties in that template while viewing this report.
        - In case of encryption or custom compression Algo etc..
    - **company_name:** Name of the client/Company owning this report
    - **storage_object_ids:** Object id in the buckets where they were stored, we require this for retrieval/ download of the reports for our clients.
    - **storage_points:**
        - Each entry here will follow the format
          Cloud-Provider - Storage Bucket Location - Type of Storage

          Example: ***gcp-euw1-standard***

        - Property represents pretty much the same details, but our client wants to delete one report specifically in one region. Then we can use this property and while fetching data we can use these details and set optimal_store_point from where we can retrieve reports fast and with less egress costs on our side.
    - **optimal_store_point:**
        - This is the Field we assign for each report based on user activity, how many times the report is being fetched and in which regions across the globe.
        - Simply, we assign this for each report based on client usage patterns over time, making the report fetching from cloud servers faster and cheaper.

Demo of Prototype Submission

- **Setting the Infra:**
  - In the folder docker setups just run **docker compose up -d** command once in mongo-db folder and minio folder.
  - Mongo-db will provide necessary databases needed for our two services i.e Template Server and report server to work.
  - Minio is a container that emulates the storage system similar to S3 and GCS, for the demo purpose of the prototype and to implement the above discussed customizations locally we used this container system to emulate GCS and AWS. When you run **docker compose up -d** in this folder, you'll find two containers getting started.
  - When all the four containers are up (assuming no conflicts while pulling images or ports being blocked), you should see this output for **docker ps** command. (added formatting for neat image)



```
ganarko@gana-mnet:~/.../dockersetups/minio$ docker ps --format "{{.Names}}: \t{{.Status}}: \t {{.Ports}}: "
minio-gcp:      Up 35 minutes:   0.0.0.0:9010->9000/tcp, :::9010->9000/tcp, 0.0.0.0:9011->9001/tcp, :::9011->9001/tcp:
minio-aws:      Up 35 minutes:   0.0.0.0:9020->9000/tcp, :::9020->9000/tcp, 0.0.0.0:9021->9001/tcp, :::9021->9001/tcp:
mongodb-admin:  Up 2 hours:      0.0.0.0:8081->8081/tcp, :::8081->8081/tcp:
mongoDB:        Up 2 hours:      0.0.0.0:27017->27017/tcp, :::27017->27017/tcp:
ganarko@gana-mnet:~/.../dockersetups/minio$ 
```

  - Please note minio containers here emulate behavior of object storage of S3 and GCS, this gives me flexibility to showcase my solution without creating accounts across multiple cloud platforms.
  - Now you can run both the micro services by going into the **file_server** directory and running the command **python3 manage.py runserver.py**
  - Now you can import and use the postman collection to test the process.
  - Create template → Generate Report (for now we just create a pdf file with give text content) → Save it to minio storage buckets (based on template properties for the associated report name)

What our Solution Achieves:

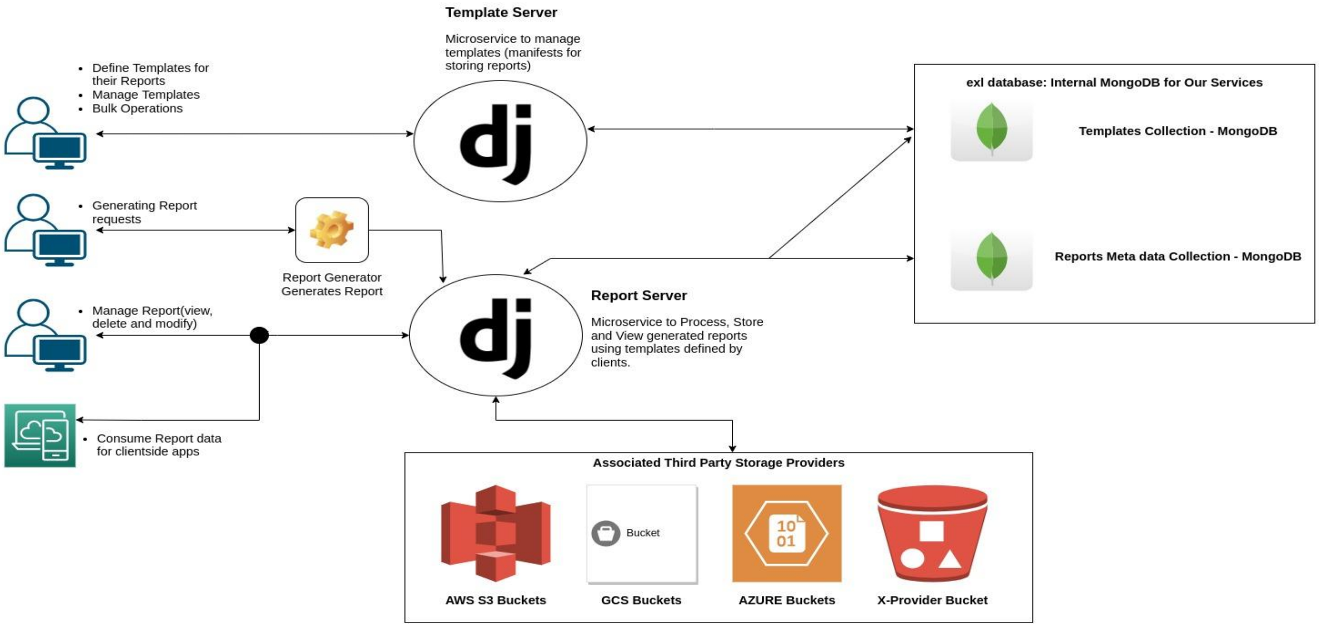
- **Client-Flexibility:** Clients can customize and use features provided by various cloud providers in terms of security and accessibility, at the same time they can take advantage of pricing models of different cloud providers to optimize their charges. [One of the Core problem for our takeaways from the problem statement]
- **Cost to EXL:** Provides us with options to optimize our infra budget by deciding optimal_store points which will cut down our egress costs.
- **Scalability:** Solution can be scalable based on our requirement as two micro services are independent of each other.
- **Ease of Operations:** Clients can group similar files in a template and can decide their properties once, very useful when the reports database for a given company gets bigger over time.
- **Multiple Cloud Providers:** As we've seen in the prototype demo, in real life scenario each minio container will represent a cloud provider, thus cloud agnostic.

Problems to be Addressed:

- **Authentication:** Use auth for APIs exposed to clients. Prefer HTTPS for end product
- **Tightly Coupled Services:** In this case we are just generating a simple pdf report on api call for demo, but practically that report generating service will be a whole other app doing the work, so how to effectively send generated reports to the report server?
    - **Possible Solution:** Mount same storage Volume for both the apps and report generator can just send an internal call to report server post generation of report.
- **Reliability:** What happens when a report fails to upload?
    - **Possible Solution:** Scheduled retries with acknowledgement
- **Collision:** In-case same file-name/pattern appeared in more than one template?
    - **Possible Solution:** Assign Priority Score for templates, so when collision happens templates with higher Score will be taken into consideration.
- **Template Change:** In-case of template modification, what will happen to already stored reports, which used previous data of the template?.
    - **Possible Solution:** implementing versioning for Templates

Note:

- Reason to use object storage like minio is that it can perfectly replicate the process of storing objects within localhost and we get to visualize customization about multiple regions and buckets easily in this model, compared to doing the same by creating multiple accounts/buckets in multiple cloud consoles.
- In-case of django/docker-compose/python version incompatibility or any other blocker, you can always see the demo-video for how the prototype operates**.**
- To further understand the architecture of the solution you can see the high level Architecture Diagram for the solution. **(*exl-design-diagram.jpg*)**


Template Server
Microservice to manage templates (manifests for storing reports)
• Define Templates for their Reports
• Manage Templates
• Bulk Operations
exl database: Internal MongoDB for Our Services
Templates Collection - MongoDB
Reports Meta data Collection - MongoDB
• Generating Report requests
Report Generator Generates Report
Report Server
Microservice to Process, Store and View generated reports using templates defined by clients.
• Manage Report(view, delete and modify)
• Consume Report data for clientside apps
Associated Third Party Storage Providers
Bucket
10 01
AWS S3 Buckets
GCS Buckets
AZURE Buckets
X-Provider Bucket

Resources:

- References for encryption:
  - How to Encrypt and Decrypt Files in Python - Python Code
  - Symmetric vs. Asymmetric Encryption - What are differences?
  - GCS Encryption
  - AWS S3 encryption
- References for compression:
  - All The Ways to Compress and Archive Files in Python | by Martin Heinz | Towards Data Science
  - python - How to compress a large file? - Stack Overflow
  - How to Compress files with ZIPFILE module in Python.
  - GCS Compression
  - AWS | S3 compression
- Docker setup: noSQL database and minio-storage set up:
  - MinIO | Guide